UIC COMPUTER SCIENCE

Project #04: Databases and AVL Trees

Complete By: Monday, October 28th @ 11:59pm
Assignment: Database program
Policy: Individual work only, late work *is* accepted (see "Policy" section for more details)
Submission: via Gradescope --- limited to 12 submissions

Background

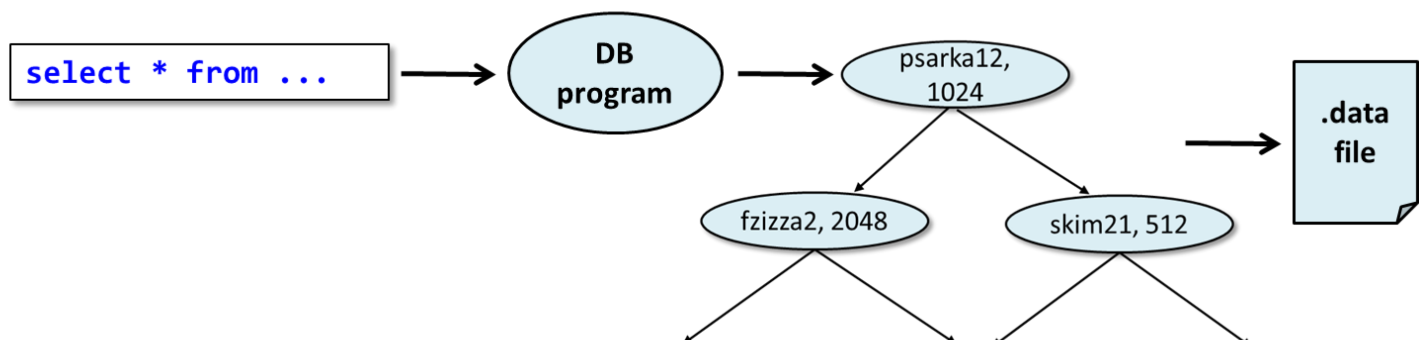
Conceptually, a **database** is a set of **tables**, where each table stores data that is logically related. For example, the database maintained by UIC would include the following tables:

students: data about each student
faculty: data about each faculty member
staff: data about each staff employee
courses: data about each course

Databases are created, searched and modified using the **SQL** programming language. For example, here's an SQL select query to retrieve all the info from the students table about the student with netid fzizza2:

```
select * from students where netid = fzizza2
```

How are databases implemented? Each table is stored in a file, and balanced trees are built to index the file and find data quickly. For example, suppose student fzizza2 has their data stored at position 2048 in the underlying file (which would be called "students.data" based on the tablename). This file would be indexed by a tree consisting of (uin, offset) pairs, and this student's node in the tree would contain the pair (fzizza2, 2048).



Your Assignment

Your assignment is to build a database-like program that inputs SQL select queries, and retrieves the requested data. Your program will build and use trees to find data that is indexed, and otherwise use linear search to find data that is not indexed. Here's an example interaction with the program:

```
C:\Classes\cs251\project04-database with avl\database\Debug\database.exe
Welcome to myDB, please enter tablename> students
Reading meta-data...
Building index tree(s)...
Index column: uin
  Tree size: 6
  Tree height: 2
Index column: netid
  Tree size: 6
  Tree height: 2

Enter query> select * from students where uin = 123456        (1)
uin: 123456
firstname: pooja
lastname: sarkar
netid: psarka12
email: pooja@piazza.com

Enter query> select email from students where netid = fzizza2   (2)
email: zizza@ucla.edu

Enter query> select netid from students where lastname = kim     (3)
netid: mkim16
netid: skim21

Enter query>
```

In this case the underlying "students" table has 2 indexes, based on columns **uin** and **netid**. Searches based on those columns are fast since they will take advantage of an underlying AVL tree; examples are select queries #1 and #2 shown above. However, select query #3 searches based on **lastname**, which is not indexed. This will require a linear search of the underlying "students.data" file.

File formats

File formats are different from what you might expect, so please read this carefully. A **table** is represented using 2 files: one that contains meta-data (information about the table), and one that contains the data itself. For simplicity, we are going to use text files, so you can open both files in a text editor and review the contents. All data will be string-based, again for simplicity. Let's look at the **students** example.

For the **students** table, the meta-data file will be called "students.meta", and the data file "students.data". Even though these are text files, note that you cannot double-click on the files to open --- since the extensions are .meta and .data, and not .txt. To open these files, you'll need to start your text editor, and then open from within the text editor program itself.

First, let's look at the data stored in "students.data". We are storing 5 values per student: uin, first name, last name, netid, and email address:

```
123456 pooja sarkar psarka12 pooja@piazza.com .................................
456789 frank zizza fzizza2 zizza@ucla.edu .....................................
789123 mee kim mkim16 mkim16@uic.edu ..........................................
238117 lucy johnson ljohns21 lucy.johnson@gmail.com ...........................
556178 drago kolar dkolar8 drago1998@yahoo.com ................................
732290 soo kim skim21 skim21@uic.edu ..........................................
```

The "." can be ignored; they are used to visually pad the line so that every student "record" in the file is the same size. All data values are treated as strings, even though they may look like integers (e.g. uin). The .data file is in Windows format, and must remain in Windows format (with \r and \n at the end of each line); do not convert these files to your local text file format.

Looking at the data as a whole, each kind of data --- uin, first name, last name, etc. --- is viewed as a **column** of data. This implies the students table contains 5 columns. Where are the names of these columns defined? Which columns are indexed to enable fast lookup? This is the type of "meta-data" stored in the 2nd file, "students.meta":

```
82
5
uin 1
firstname 0
lastname 0
netid 1
email 0
```

The first line is the **record size RS**, and denotes the amount of data (in bytes) stored per student. This implies the first student will be stored at offset 0, the 2nd student at offset 82, the 3rd student at 164, and so on. In general, record offsets are a multiple of RS. The second line is the **# of columns C** in the table, and corresponds to the number of data values stored per record. For example, each student record contains 5 data values (which matches what we saw earlier in "students.data"). Lastly, the remainder of the .meta file contains C lines, one per column, defining the name of the column along with whether this column is indexed (1) or not (0). Given the "students.meta" file shown above, we know the students table has 5 columns --- uin, firstname, lastname, netid, email --- where **uin** and **netid** are indexed. Assume that the order of the column names in .meta file matches the order of the data in each record of the .data file. In other words, in "students.data", the 1st data value is the uin, the 2nd data value is the firstname, the 3rd data value is the lastname, and so on.

In terms of C++, the .meta file can be opened and processed as you might expect: use an **ifstream** object, call **file.good( )** to make sure it was opened successfully, and input the data using the **>>** operator. In general, since the # of columns is unpredictable, you'll want to save the meta-data in a data structure such as vector.

The .data file requires different processing because it's based on fixed-sized records. The use of fixed-sized records is necessary so that data for a particular student can be found quickly --- by jumping to a particular offset (versus reading the file line by line). You open the file as a **binary file**, and you move from line to line by moving from the start of one record to the start of the next. Here's an example of opening a .data file (such as "students.data") and echoing the values stored in each record:

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
#include <sstream>

using namespace std;

void EchoData(string tablename, int recordSize, int numColumns)
{
  string   filename = tablename + ".data";
  ifstream data(filename, ios::in | ios::binary);

  if (!data.good())
  {
     cout << "**Error: couldn't open data file '" << filename << "'." << endl;
     return;
  }

  //
  // Okay, read file record by record, and output each record of values:
  //
  data.seekg(0, data.end);  // move to the end to get length of file:
  streamoff length = data.tellg();

  streamoff pos = 0;  // first record at offset 0:
  string    value;

  while (pos < length)
  {
     data.seekg(pos, data.beg);  // move to start of record:

     for (int i = 0; i < numColumns; ++i)  // read values, one per column:
     {
       data >> value;
       cout << value << " ";
     }

     cout << endl;
     pos += recordSize;  // move offset to start of next record:
  }
}
```

To echo the data in "students.data", you would call as follows: **`EchoData("students", 82, 5);`**

Assignment details

In a database, the data stays in the .data file. Trees are used to find the data quickly, but the (key, value) pairs stored in the tree are of the form (key, record offset) --- not (key, data). When the program starts, you'll read the .meta file and determine which columns of data are indexed. For each index, you'll build a tree. How is the tree built? You open the .data file, read through the data record by record, and insert into the tree the **(key, offset)** pair for that record. You don't insert the data into the tree, you insert the record positions so you can find the data later. This is how databases work, and how you must approach this assignment.

For example, the **students** meta-data states that the **uin** and **netid** columns are indexed. Using the "students.data" file we saw earlier:

```
123456 pooja sarkar psarka12 pooja@piazza.com ..................................
456789 frank zizza fzizza2 zizza@ucla.edu ......................................
789123 mee kim mkim16 mkim16@uic.edu ...........................................
.
.
```

To build the **uin** index tree, you would insert ("123456", 0), then ("456789", 82), then ("789123", 164), etc. The offsets are multiples of the record size, which is 82 in this case. To build the **netid** index tree, you would insert ("psarka12", 0), then ("fzizza2", 82), then ("mkim16", 164), etc. The keys are strings, and the record offsets are type **streamoff** defined in #include <fstream>. This implies your trees should be of type avltree<string, streamoff>.

One of the more interesting aspects of the assignment is that you need to use other data structures. There can be any number of columns, and any number of indexes. So you'll need to keep track of the columns, and your index trees, using a data structure. [*Hint: std::vector is a good choice.*]

Use functions. For example, I found the following function very helpful for reading a record from the .data file. Here's the design, we'll leave the coding details to you:

```
//
// GetRecord
//
// Reads a record of data values and returns these values in a vector.
// Pass the table name, the file position (a stream offset), and the #
// of columns per record.
//
// Example:  GetRecord("students", 0, 5) would read the first student
// record in "students.data".
//
vector<string> GetRecord(string tablename, streamoff pos, int numcolumns)
{
   // open the file, make sure it opened, seekg to the given position,
   // loop and input values using >>, store into vector, return vector
}
```

This function opens and closes the file every time; this may not be the most efficient approach, but it's a good place to start. If time permits, you can improve efficiency by opening the .data file once in main(), leaving it

open, and passing the file object instead of the tablename; you'll need to pass the file object by reference.

Use the avltree class you built in project #03; a compiled avl.o object file will be provided on Codio if you were unable to complete project #03. However, do not change the tree in any significant ways, the current plan for testing submissions is to run your database program using our avltree class so we can report tree usage statistics; this implies you cannot change the public API. That said, we ran into some compilation errors around the avltree class when returning trees from a function, so you might need to modify the copy constructor declaration by adding the **const** keyword as shown below:

```
// copy constructor:
avltree(const avltree& other)
{
   Root = nullptr;
   Size = 0;

   _copytree(other.Root);
}
```

Likewise, in some cases you might want to assign a tree into another variable, which can also happen when returning a tree from a function. In this case you need to overload the assignment operator **=**, like this:

```
avltree& operator=(const avltree& other)
{
   clear();

   _copytree(other.Root);

   return *this;
}
```

Programming Environment and Getting Started

You are free to program in any environment you want; final submissions will be collected using Gradescope. If you want to use Codio, a project has been created named "**cs251-project04-myDB**". The environment will contain catch and valgrind for testing purposes.

A skeleton "main.cpp" file is provided that contains the EchoData function shown earlier, as well as some code that implements the main loop. Type "**make build**" to compile, and "**make run**" to run. Sample input files are provided as well.

The makefile is setup to compile and run against our implementation of the avltree class. If you would prefer to use your own avltree class, edit the makefile and remove all references to "avl.o". Then delete the "avl.o" and "avl.h" files, and install your own version of "avl.h".

Requirements

Design and build a database program as outlined in the previous sections. You must build an AVL tree for each indexed column, and use that tree for fast lookup when the select query's **where** clause references that column. Here's another screenshot --- the **uin** index tree must be used in processing queries #1 and #2:

Microsoft Visual Studio Debug Console

```
Welcome to myDB, please enter tablename> students
Reading meta-data...
Building index tree(s)...
Index column: uin
  Tree size: 6
  Tree height: 2
Index column: netid
  Tree size: 6
  Tree height: 2

Enter query> select * from students where uin = 556177   (1)
Not found...

Enter query> select * from students where uin = 556178   (2)
uin: 556178
firstname: drago
lastname: kolar
netid: dkolar8
email: drago1998@yahoo.com

Enter query> select * from students where firstname = dale   (3)
Not found...

Enter query> select * from students where firstname = soo   (4)
uin: 732290
firstname: soo
lastname: kim
netid: skim21
email: skim21@uic.edu

Enter query> exit
```

For queries #3 and #4 shown above, linear search of the underlying .data file must be performed since **firstname** is not an indexed column.

Reasonable error checking should be performed, in particular around the user's input. If the user enters a query other than select or exit, it's an error. A valid select query has 8 components. Version 1 selects a particular column of data, given as columnname1 in the query below:

```
select columnname1 from tablename where columnname2 = value
```

Version 2 selects all columns of data by replacing columnname1 with *:

```
select * from tablename where columnname2 = value
```

For ease of input processing, assume all query components are separated by a single space. Here's a

screenshot of some error handling cases:

```
Enter query> insert ...
Unknown query, ignored...

Enter query> select gpa from students where uin = 123456
Invalid select column, ignored...

Enter query> select netid from students where gpa = 3.5
Invalid where column, ignored...

Enter query> select * but I forget the rest
Invalid select query, ignored...

Enter query> select netid from table where uin = 123456
Invalid table name, ignored...

Enter query>
```

Program submission

Submit your program files to Gradescope for grading under **Project 04 - myDB**. Note that your program will be compiled and run against our version of the avltree class; this is necessary to collect stats about tree usage for grading.

Your work will receive a tentative grade for correctness, and then will undergo manual review for commenting, readability, approach, and adherence to requirements. Note that you'll be limited to a total of 12 submissions; your tentative score will be the highest value earned from your submissions.

Commenting/readability/approach will count for 10% of project score. Adherence to requirements may count anywhere from 10% to 100%.

Policy

Late work *is* accepted. You may submit as late as 24 hours after the deadline for a penalty of 10%. After 24 hours, no submissions will be accepted.

All work submitted for grading *must* be done individually. While we encourage you to talk to your peers and learn from them (e.g. your "iClicker teammates"), this interaction must be superficial with regards to all work submitted for grading. This means you *cannot* work in teams, you cannot work side-by-side, you cannot submit someone else's

work (partial or complete) as your own. The University's policy is available here:

https://dos.uic.edu/conductforstudents.shtml .

In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. It is also considered academic dishonesty if you click someone else's iClicker with the intent of answering for that student, whether for a quiz, exam, or class participation. Academic dishonesty is unacceptable, and penalties range from a letter grade drop to expulsion from the university; cases are handled via the official student conduct process described at https://dos.uic.edu/conductforstudents.shtml .